Potential-aware Automated Abstraction of Sequential Games, and Holistic Equilibrium Analysis of Texas Hold'em Poker*

Andrew Gilpin
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
gilpin@cs.cmu.edu

Tuomas Sandholm
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
sandholm@cs.cmu.edu

Troels Bjerre Sørensen
Department of Computer Science
University of Aarhus
Åbogade 34, Århus, Denmark
trold@daimi.au.dk

Abstract

We present a new abstraction algorithm for sequential imperfect information games. While most prior abstraction algorithms employ a myopic expected-value computation as a similarity metric, our algorithm considers a higher-dimensional space consisting of histograms over abstracted classes of states from later stages of the game. This enables our bottom-up abstraction algorithm to automatically take into account *potential*: a hand can become relatively better (or worse) over time and the strength of different hands can get resolved earlier or later in the game. We further improve the abstraction quality by making multiple passes over the abstraction, enabling the algorithm to narrow the scope of analysis to information that is relevant given abstraction decisions made for earlier parts of the game. We also present a custom indexing scheme based on suit isomorphisms that enables one to work on significantly larger models than before.

We apply the techniques to heads-up limit Texas Hold'em poker. Whereas all prior game theory-based work for Texas Hold'em poker used generic off-the-shelf linear program solvers for the equilibrium analysis of the abstracted game, we make use of a recently developed algorithm based on the *excessive gap technique* from convex optimization. This paper is, to our knowledge, the first to abstract and game-theoretically analyze all four betting rounds in one run (rather than splitting the game into phases). The resulting player, *GS3*, beats *BluffBot*, *GS2*, *Hyperborean*, *Monash-BPP*, *Sparbot*, *Teddy*, and *Vexbot*, each with statistical significance. To our knowledge, those competitors are the best prior programs for the game.

Introduction

Automatically determining effective strategies in stochastic environments with hidden information is an important and difficult problem. In multiagent systems, the problem is exacerbated because the outcome for each agent depends on the strategies of the other agents. Poker games are well-defined environments exhibiting many challenging properties, including adversarial competition, uncertainty (with respect to the cards the opponent currently holds), and stochasticity (with respect to the uncertain future card deals). Poker games been identified as an important testbed for research on these topics (Billings *et al.* 2002). Consequently, many researchers have chosen poker as an application area in which to test new techniques. In particular, heads-up limit Texas Hold'em poker has recently received a large amount of research attention, *e.g.*, (Korb, Nicholson, & Jitnah 1999; Billings *et al.* 2002; 2003; 2004; Gilpin & Sandholm 2006a; 2007). It can be modeled as a two-person zero-sum game, which has both strategic and computational implications.

From a strategic perspective, two-person zero-sum games are attractive because the set of Nash equilibria for these games are *interchangeable* and offer a *guaranteed security level*. The interchangeable property states that if (x, y) is a Nash equilibrium (where x is player 1's mixed strategy and y is player 2's mixed strategy) and (x', y') is a Nash equilibrium, then (x, y') and (x', y) are also Nash equilibria. Among other things, this eliminates the *equilibrium selection problem*, which occurs in some games where there are multiple equilibria. The guaranteed security level means that by playing a Nash equilibrium strategy, a player is guaranteed a certain minimum expected payoff, *regardless of the strategy used by the other player*. In two-person zero-sum games, the value that one player can guarantee is the negative of what the other player can guarantee.

From a computational perspective, two-person zero-sum games have the benefit that Nash equilibria can be computed in time polynomial in the size of the game description. In particular, the equilibrium problem can be modeled and solved as a linear program (LP) (Romanovskii 1962; Koller & Megiddo 1992; von Stengel 1996).

Although equilibrium strategies for two-person zero-sum games can be computed efficiently in theory, there are two important reasons why new techniques are still needed to enable the application of game theory to large problems, such as poker. The first is that the games themselves are huge. For example, heads-up limit Texas Hold'em has a game tree with around 10^{18} nodes. Even explicitly representing this game would require an enormous (impractical) amount of memory. The second reason is that even in cases where a game can be represented in memory (for example, after abstracting the game to find a smaller, almost equivalent representation), the LP solvers that are currently fastest for these problems (CPLEX's interior-point method) require an amount of memory that is several orders of magnitude larger than the representation of the game (Gilpin & Sandholm 2006b).

Most existing approaches handle these two problems by abstraction and splitting the game into two phases. These cause strategic errors. Our player differs from prior approaches along both of these lines. First, the abstraction in

*This material is based upon work supported by the National Science Foundation under ITR grant IIS-0427858.

the prior approaches is typically crafted manually (Billings *et al.* 2003) or by myopic algorithms (Gilpin & Sandholm 2006a; 2007). In this paper, we develop a non-myopic abstraction algorithm that addresses not only the winning probability but also the potential. Second, we do not split the game into phases; instead, we tackle the entire four-round model holistically in one single optimization. Such scalability of equilibrium finding is made possible by our application of the *excessive gap technique* (Nesterov 2005), which was recently specialized to equilibrium finding in two-person zero-sum sequential imperfect information games (Hoda, Gilpin, & Peña 2006).

Rules of the game

There are many variations of poker. Like most prior work, we focus on two-player (heads-up) limit Texas Hold'em. The rules are as follows. The *small bet* is two chips. Before any cards are dealt, the first player (*i.e.,* small blind) puts one chip into the pot, and the second player (*i.e.,* big blind) puts two chips into the pot. There are four betting rounds. In the first, each player is dealt two cards, face down (these are called the *hole cards*). The small blind may either *call* the big blind (add one chip to the pot), *raise* (three chips), or *fold* (zero chips). The players then alternate either *calling* the current bet (contributing two chips), *raising* the bet (four chips), or *folding* (zero chips). In the event of a fold, the folding player forfeits the game and the other player wins all of the chips in the pot. Once a player calls a bet, the betting round finishes. The number of raises is limited to four in each round. In the second, third, and fourth rounds, three, one, and one *community cards* are dealt face up, respectively. In each of these rounds the big blind acts first; betting proceeds as in the first round. The bets in the last two rounds are twice as large as in the first two rounds. If the final round ends with neither player folding, the player who forms the best five-card hand using any of his two cards and the five community cards wins the chips in the pot; in the event of a tie, the players split the pot.

Prior programs for Texas Hold'em poker

There has been a recent surge of research into developing effective computer programs for playing heads-up limit Texas Hold'em. We now describe these prior approaches.

The first successful game theory-based player for Texas Hold'em was constructed by modeling the game as two phases. For each phase, a domain expert manually designed a coarse abstraction, which was then solved as an LP using an interior-point method. The player is competitive with advanced human players (Billings *et al.* 2003). A player based on these techniques is available in the commercial software package *Poker Academy Pro* as *Sparbot*.

Opponent modeling is a technique in which a program attempts to identify and exploit opponents' weaknesses (Billings *et al.* 2004; Sturtevant, Zinkevich, & Bowling 2006). This can be done by building a model for predicting opponents' actions based on observations made throughout game play. The most successful Texas Hold'em program from that line of research is *Vexbot* (Billings *et al.* 2004). It combines opponent modeling with miximax search (a variant of minimax search, which allows the players to move probabilistically according to some model to account for the presence of imperfect information).

Recently, the game theory-based player *GS1* was presented, which featured automated abstraction and real-time equilibrium approximation (Gilpin & Sandholm 2006a). The abstraction algorithm used for that player was a simple approximation version of the *GameShrink* algorithm (Gilpin & Sandholm 2006b). *GS1* is competitive with *Sparbot* and *Vexbot*, but there is no statistically significant evidence to demonstrate that it is better or worse than them. Recently, the authors of *GS1* introduced an improved abstraction algorithm and a method for computing leaf payoffs of truncated games, which led to the newer player *GS2* (Gilpin & Sandholm 2007), which was shown to be better than *GS1* (by a statistically significant margin) and competitive with *Sparbot* and *Vexbot*.

The first AAAI Computer Poker Competition was held in 2006 (Littman & Zinkevich 2006). There were two competitions: the *Bankroll Competition* and the *Series Competition*. The Bankroll Competition determined the winner based on which player won the most money overall (thus emphasizing the exploitation of weak opponents), whereas the Series Competition determined the winner based on who beat the most opponents (thus emphasizing strong players that cannot be easily exploited). Given this, the Series Competition is the most relevant to our research goal of developing strong unbeatable game-theoretic agents.

The first and second place winners of that competition were *Hyperborean* and *BluffBot*, respectively. (*GS2*, discussed above, came in third place.) Although detailed information about these two players is not publicly available, it is known that both are based on game-theoretic techniques. *Hyperborean* is similar to *Sparbot*; both were developed by the University of Alberta Computer Poker Research Group. *BluffBot* was developed by Teppo Salonen, and is described as "a combination of an expert system and a game-theoretic pseudo-optimal strategy."[1] Another competitor in that competition was *Monash-BPP* (Korb, Nicholson, & Jitnah 1999), which is based on Bayesian networks for modeling the player's hand, the opponent's hand, and the opponent's strategy (conditional on its hand). The final program in the competition was *Teddy*, developed by Morten Lynge. To our knowledge, there is no information on this program publicly available. Later in this paper we will present experiments against each of these prior programs.

All of the players described above are for (non-tournament) heads-up *limit* Texas Hold'em. Another form of Texas Hold'em is a *no-limit tournament*. In a poker tournament, each player begins with the same number of chips, and poker is played repeatedly until only one player has chips left. In no-limit poker, each player can place bets with sizes up to the amount of chips they have left. Recently, near-optimal strategies for the later stages of a no-limit tournament were computed (Miltersen & Sørensen 2007). However, the results of that paper are not comparable to the

[1] `http://www.bluffbot.com/`

present work due to the differences in the game being studied. For one, their recommended strategy relies heavily on the player being allowed to bet all of his remaining chips.

Overview of our approach

To construct our player, we first compute a four-round abstraction of the Texas Hold'em state space. Our abstraction is constructed so that the resulting size of the state-space is manageable by our equilibrium approximation algorithm. We discuss the abstraction construction in the next two sections. After that, we discuss how our algorithms take advantage of suit isomorphisms to speed up running time and decrease memory requirements.

Once the abstraction is computed, we run an algorithm for finding an approximate equilibrium in the abstracted game. The resulting strategies (once mapped back into the original game) represent our player, *GS3*.

Deciding the coarseness of the abstraction

Before computing an abstraction, we need to decide how coarse an abstraction we want. Ideally, we would compute an abstraction as fine-grained as possible. However, we need to limit the fineness of the abstraction to ensure that we are able to compute an equilibrium approximation for the resulting abstracted game.

One important aspect of the abstraction is the *branching factor*. One intuitively desirable property is to have an abstraction where the relative amount of information revealed in each stage is similar to the relative amount revealed in the game under consideration. For example, it would likely not be effective to have an abstraction that only had one bucket for each of the first three rounds, but had 1000 buckets for the last round. Similarly, we don't want to have 100 buckets in the first round if we are going to only have 100 buckets in the second, third, and fourth rounds, since then no new information would be revealed after the first round.

One implication of this reasoning is that the branching factor going into the flop (where three cards are dealt) should be greater than the branching factor going into the turn or river (where only one card is dealt in each round). Furthermore, it seems reasonable to require that the branching factor of the flop be at least the branching factor of the turn and river *combined*, since more information is revealed on the flop than on the turn and river together.

Based on these considerations, and based on some preliminary experiments to determine the problem size we could expect our equilibrium-finding algorithm to handle, we settled on an abstraction that has 20 buckets in the first round, 800 buckets in the second round, 4,800 buckets in the third round, and 28,800 buckets in the fourth round.[2] This implies a branching factor of 20 for the pre-flop, 40 for the flop, 6 for the turn, and 6 for the river.

[2]To enable the solving for equilibrium with such fine-grained abstraction, we model the game as having at most three raises per betting round instead of four. This approach is commonly adopted when building computer programs for playing poker (Billings *et al.* 2003; Gilpin & Sandholm 2006a; 2007). In practice, play very seldom proceeds to a fourth raise anyway.

Potential-aware automated abstraction

The most successful prior approach to automated abstraction in sequential games of imperfect information was based on a myopic expected-value computation (Gilpin & Sandholm 2007), and used k-means clustering with integer programming to compute the abstraction. A state of the game was evaluated according to the probability of winning the hand. The algorithm clustered together states with similar probabilities of winning, and it started computing the abstraction from the first round and then down through the card tree. This *top-down* algorithm generated the abstraction for *GS2*.

That approach does not take into account the *potential* of hands. For instance, certain poker hands are considered *drawing hands* in which the hand is currently weak, but has a chance of becoming very strong. A common example of a drawing hand is one in which the player has four cards of the same suit (five are required to make a *flush*);[3] at the present stage the hand is not very strong, but could become so if a card of the same suit showed up later in the game. Since the strength of such a hand could potentially turn out to be much different later in the game, it is generally accepted among poker experts that such a hand should be played differently than another hand with a similar chance of winning, but without as much potential (Sklansky 1999).[4] However, if using the difference between probabilities of winning as the clustering metric, the abstraction algorithm would consider these two very different situations similar.

One possible approach to handling the problem that certain hands with the same probability of winning may have different potential would be to consider not only the expected strength of a hand, but also its variance. In other words, the algorithm would be able to differentiate between two hands that have the same probability of winning, but where one hand faces more uncertainty about its final strength. Although this would likely be an improvement over basic expectations-based abstraction, it fails to capture two important issues that prevail in many sequential imperfect information games, including poker:

- Mean and variance are a lossy representation of a probability distribution, and the lost aspects of the probability distribution over hand strength can be significant for deciding how one should play in any given situation.
- The approach based on mean and variance does not take into account the different *paths of information revelation* that hands take in increasing or decreasing in strength. For example, two hands could have similar means and variances, but one hand may get the bulk of its uncertainty re-

[3]The rules of poker define the rank (relative strength) of different hands. See, *e.g.*, `http://www.pagat.com/vying/pokerrank.html`.

[4]In the manual abstraction used in *Sparbot*, there are six buckets of hands where the hands are selected based on likelihood of winning and one extra bucket for hands that an expert considered to have high potential (Billings *et al.* 2003). In contrast, our approach is automated, and does its bucketing holistically based on a multi-dimensional notion of potential (so it does not separate buckets into ones based on winning probability and ones based on potential). Furthermore, its abstraction is drastically finer grained.

solved in the next round, while the other hand needs two more rounds before the bulk of its final strength is determined. The former hand is better because the player has to pay less to find out the essential strength of his hand.

To address these issues, we instead introduce an approach where we associate with each state of the game a *histogram* over future possible states. This representation can encode all the pertinent information from the rest of the game (such as paths of information revelation), unlike the approach based on mean and variance. As in prior automated abstraction approaches, the (k-means) clustering step requires a distance function to measure the dissimilarity between different states. The metric we use in this paper is L_2-distance. Specifically, let $\mathcal{S}$ be a finite set of future states, and let each hand i be associated with a histogram, h_i, over the future states $\mathcal{S}$. Then, the distance between hands i and j is $dist(i,j) = \left[\sum_{s \in \mathcal{S}} \left(h_i(s) - h_j(s)\right)^2\right]^{\frac{1}{2}}$.

There are at least two prohibitive problems with this vanilla approach as stated. First, there are a huge number of possible reachable future states, so the dimensionality of the histograms is too large to do meaningful clustering with a reasonable number of clusters (*i.e.,* small enough to lead to an abstracted game that can be solved for equilibrium). Second, for any two states at the same level of the game, the descendant states are disjoint. Thus the histograms would have non-overlapping supports, so any two states would have maximum dissimilarity and thus no basis for clustering.

For both of these reasons (and for reducing memory usage and enhancing speed), we coarsen the domains of the histograms. First, instead of having histograms over individual states, we use histograms over abstracted states (clusters), which contain a number of states each. We will have, for each cluster, a histogram over clusters later in the game. Second, we restrict the histogram of each cluster to be over clusters at the next level of the game tree only (rather than over clusters at all future levels). However, we introduce a technique (a bottom-up pass of constructing abstractions up the tree) that allows the clusters at the next level to capture information from all later levels.

One way of constructing the histograms would be to perform a bottom-up pass of a tree representing the possible card deals: abstracting level four (*i.e.,* betting round 4) first, creating histograms for level 3 nodes based on the level 4 clusters, then abstracting level 3, creating histograms for level 2 nodes based on the level 3 clusters, and so on. This is indeed what we do to find the abstraction for level 1.

However, for later betting rounds, we improve on this algorithm further by leveraging our knowledge of the fact that abstracted children of any cluster at the level above should only include states that can actually be children of the states in that cluster. We do this by *multiple* bottom-up passes, one for each cluster at the level above. For example, if a cluster at level 1 contains only those states where the hand consists of two Aces, then when we are doing abstraction for level 2, the bottom-up pass for that level-1 cluster should only consider future states where the hand contains two Aces as the hole cards. This enables the abstraction algorithm to narrow the scope of analysis to information that is relevant given the abstraction that it made for earlier levels. The following subsections describe our abstraction algorithm in detail.[5]

Computing the abstraction for round 1

The first piece of the abstraction we computed was for the first round, *i.e.,* the pre-flop. In this round we have a target of 20 buckets, out of the $\binom{52}{2} = 1{,}326$ possible combinations of cards. As discussed above, we will have, for each pair of hole cards, a histogram over clusters of cards at level 2. (These clusters are not necessarily the same that we will eventually use in the abstraction for level 2, discussed later.)

To obtain the level-2 clusters, we perform a bottom-up pass of the card tree as follows. Starting with the fourth round, we cluster the $\binom{52}{2}\binom{50}{5} = 2{,}809{,}475{,}760$ hands into 5 clusters[6] based on the probability of winning. Next, we consider the $\binom{52}{2}\binom{50}{4} = 305{,}377{,}800$ third-round hands. For each hand we compute its histogram over the 5 level-4 clusters we computed. Then, we perform k-means clustering on these histograms to identify 10 level-3 clusters. We repeat a similar procedure for the $\binom{52}{2}\binom{50}{3} = 25{,}989{,}600$ hands in the second round to identify 20 level-2 clusters.

Using those level-2 clusters, we compute the 20-dimensional histograms for each of the $\binom{52}{2} = 1{,}326$ possible hands at level 1 (*i.e.,* in the first betting round). Then we perform k-means clustering on these histograms to obtain the 20 buckets that constitute our abstraction for the first betting round.

Computing the abstraction for rounds 2 and 3

Just as we did in computing the abstraction for the first round, we start by performing a bottom-up clustering, beginning in the fourth round. However, instead of doing this bottom-up pass once, we do it once for each bucket in the first round. Thus, instead of considering all $\binom{52}{2}\binom{50}{5} =$ 2,809,475,760 hands in each pass, we only consider those hands that contain as the hole cards those pairs that exist in the particular first-round bucket we are looking at.

At this point we have, for each first-round bucket, a set of second-round clusters. For each first-round bucket, we have to determine how many child buckets it should actually have. For each first-round bucket, we run k-means clustering on its second-round clusters for $k \in \{1..80\}$. (In other words, we are clustering those second-round clusters (*i.e.,* data points) into k clusters.) This yields, for each first-round bucket and

[5] If no limit is imposed on the fineness of the abstraction (number of clusters of states at each level of the game), then our algorithm finds a lossless abstraction (at least if the subroutine for k-means clustering returned optimal answers). In other words, every equilibrium of that abstracted game corresponds to some equilibrium in the original game. We view this property as a necessary condition (doing the right thing in the limit) of any sensible abstraction algorithm. As such, it serves as a “sanity check”.

[6] For this algorithm, the number of clusters at each level (5 at level 4, 10 at level 3, and 20 at level 2) was chosen to honor the constraint that when clustering data, the number of clusters needed to represent meaningful information should be at least the level of dimensionality of the data. So, the number of clusters on level r should be at least as great as on level $r + 1$.

each value of k, an error measure for that bucket assuming it will have k children. (The error is the sum of each data point's L_2 distance from the centroid of its assigned cluster, weighted by the probability of each data point occurring; in poker these probabilities are all equal because cards are dealt uniformly at random.)

Based on our design of the coarseness of the abstraction, we know that we have a total limit of 800 children (*i.e.,* buckets at level 2) to be spread across the 20 first-round buckets. As in the abstraction algorithm used by *GS2* (Gilpin & Sandholm 2007), we formulate and solve an integer program (IP) to determine how many children each first-round bucket should have (*i.e.,* what k should be for that bucket). The IP simply minimizes the sum of the errors of the level-1 buckets (weighted by the probability of reaching each bucket) under the constraint that their k-values do not sum to more than 800. (The optimal k-value for different level-1 buckets varied between 18 and 51.) This determines the final bucketing for the second betting round.

The bucketing for the third betting round is computed analogously. We use level-2 buckets as the starting point (instead of level-1 buckets), and in the integer program we allow a total of 4,800 buckets for the third betting round. (The optimal k-value for different level-2 buckets varied between 1 and 10.)

Computing the abstraction for round 4

In round 4 there is no need to use the sophisticated clustering techniques discussed above since the players will not receive any more information, that is, there is no potential. Instead, we simply compute the fourth-round abstraction based on each hand's probability of winning, exactly the way as was done for computing the abstraction for *GS2* (Gilpin & Sandholm 2007). Specifically, for each third-round bucket, we consider all possible rollouts of the fourth round. Each of them constitutes a data point (whose value is computed as the probability of winning plus half the probability of tying), and we run k-means clustering on them for $k \in \{1..18\}$. (The optimal k-value for different level-3 buckets varied between 1 and 14.) The error, for each third-round bucket and each k, is the sum over the bucket's data points, of the data point's L_2 distance from the centroid of its cluster. (In general, the data points would be again weighted by their probabilities, but in poker they are all equal.)

Finally, we run an IP to decide the k for each third-round bucket, with the objective of minimizing the sum of the third-round buckets' errors (weighted by the probability of reaching each bucket) under the constraint that the sum of those buckets' k-values does not exceed 28,800 (which is the number of buckets allowed for the fourth betting round, as discussed earlier). This determines the final bucketing for the fourth betting round.[7]

Exploiting suit isomorphisms

We now introduce ways in which we exploit suit isomorphisms. We first discuss a custom indexing scheme which dramatically reduces the space requirements of representing the abstraction. In the subsection after that, we present a way to exploit suit isomorphisms to speed up a key computation.

Indexing for efficient abstraction representation

One challenge that is especially difficult when using a four-round model is that the number of distinct hands a player can face is huge. Our algorithm requires an integer index for each distinct hand in order to perform the lookup to see which abstracted bucket the given hand belongs to. The number of distinct hands, $\binom{52}{2} \cdot \binom{50}{3} \cdot \binom{47}{1} \cdot \binom{46}{1} \approx 5.6 \cdot 10^{10}$, is an order of magnitude too big to give each hand a unique index. For example, encoding the bucket for each hand using two bytes requires more than 104 gigabytes of storage. This would severely limit the practicality of the approach, since this storage is also required by our player at run-time.

We therefore introduce a more compact representation of the abstraction that capitalizes on a canonical representation of each hand based on suit symmetries. (This technique is valid since the rules of poker state that all suits are equally strong.) These canonical representations are computed using *permutations* (total orderings of the suits) and *partial permutations* (partial orderings of the suits), as we will describe later in this section.

The best size reduction one could hope for with this approach is a factor of $4! = 24$, since we can map any permutation of the four suits to the same canonical hand. That this is not fully achievable is due to the fact that some hands are unaffected by some of the permutations of the suits, *e.g.* $4\clubsuit 4\heartsuit$ is equivalent to $4\heartsuit 4\clubsuit$, in which case there are less than 24 distinct hands mapping to the same canonical one. We call this phenomenon *self symmetry*.

Our approach uses the following concept. The *colexicographical index* (Bollobás 1986) of a set of integers $x = \{x_1..x_k\} \subset \{0..n-1\}$, with $x_i < x_j$ whenever $i < j$, is $colex(x) = \sum_{i=1}^{k} \binom{x_i}{i}$. This index has the important property that for a given n, each of the $\binom{n}{k}$ sets of size k has a distinct colexicographical index. Furthermore, these indices are compactly encoded as the integers from 0 to $\binom{n}{k} - 1$.

We need to compute indices for hands from each of the four rounds. We compute these indices incrementally, using the index from round i to compute the index in round $i+1$. This approach gradually computes the permutations that map the given hand to its canonical representation. This incremental computation is useful both for providing a convenient way of computing the indices and for speeding up the index computation.

The index for the first round is computed, of course, using only the hole cards. If they are of the same suit, *e.g.* $A\clubsuit 7\clubsuit$, that suit is named "suit 1", and we get the partial permutation $\clubsuit < \{\spadesuit, \heartsuit, \diamondsuit\}$. If they are of different suits and different

[7] As discussed, our overall technique optimizes the abstraction one betting round at a time. A better abstraction could conceivably be obtained by optimizing all rounds together. However, that seems infeasible. First, the optimization problem would be nonlinear because the probabilities at a given level depend on the abstraction at all previous levels of the tree. Second, the number of decision variables in the problem would be exponential in the size of the card tree (even if the number of abstraction classes for each level is fixed). Third, one would have to solve a k-means clustering problem for each of those variables to determine its coefficient in the optimization problem.

values, *e.g.* $A\clubsuit 7\spadesuit$, we name the suit of the card with the highest value "suit 1" and the other "suit 2", resulting in the partial permutation $\clubsuit < \spadesuit < \{\heartsuit, \diamondsuit\}$. Lastly, if they have the same value, *e.g.* $7\spadesuit 7\heartsuit$, the hand is *self symmetric*, and we have the partial permutation $\{\spadesuit, \heartsuit\} < \{\clubsuit, \diamondsuit\}$. (At this point it is unspecified which of $\spadesuit$ and $\heartsuit$ is "suit 1" and which is "suit 2".)

The later rounds also give rise to partial permutations, which are then used to refine the permutation of suits that were undecided in previous rounds. For instance if the hole cards are $7\spadesuit 7\heartsuit$ and the flop is $3\diamondsuit J\diamondsuit A\heartsuit$, we refine the partial permutation $\{\spadesuit, \heartsuit\} < \{\clubsuit, \diamondsuit\}$ with $\diamondsuit < \heartsuit < \{\spadesuit, \clubsuit\}$ to get $\{\heartsuit < \spadesuit\} < \{\diamondsuit < \clubsuit\}$, *i.e.*, $\heartsuit < \spadesuit < \diamondsuit < \clubsuit$.

Then, to compute the index from the (perhaps partial) permutation, our algorithm uses a case analysis which has far too many cases (60) to describe here. As an example, if the hole cards are $7\spadesuit 7\heartsuit$ and the flop is $3\diamondsuit J\diamondsuit A\heartsuit$, then the analysis is in the category of one new suit in two cards and one old suit in a single card, breaking the self symmetry from the previous round. In this case the card with the old suit ($\heartsuit$) only has 12 possible *canonical* values (even though there are 24 $\spadesuit$s and $\heartsuit$s left in the deck), since no matter whether that new card would have been a $\spadesuit$ or a $\heartsuit$, its suit will now have become "suit 1". In the same way, the two other cards only have $\binom{13}{2}$ possible canonical values, since their suit will now have become "suit 3" no matter whether it is $\diamondsuit$ or $\clubsuit$. Thus this case has $\binom{13}{2} \cdot 12 = 936$ canonical hands representing four times that many actual hands, all of which share $7\spadesuit 7\heartsuit$ as the hole cards.

Each of the cases simply breaks the hand up into sets to be encoded with colexicographical indexing. In the case above, the sets are $\{1{,}9\}$ and $\{11\}$ (index 11 is the highest of the 12 cards in the group). Here, 1 corresponds to the three, 9 corresponds to the Jack, and 11 corresponds to an Ace. The index within this case is then computed as $colex(\{1, 9\}) \cdot 12 + colex(\{11\})$. This is then combined with the index within the case of the first round. There are 13 possible pairs, and our sevens have index 5. We thus get $(colex(\{1, 9\}) \cdot 12 + colex(\{11\})) \cdot 13 + 5$. Finally, to get the index, this is added to a global offset associated with this particular case.

With this indexing scheme, the memory consumption of the index for all four rounds reduces by a factor of 23.1 (which is close to the optimistic upper bound of 24).

Using symmetries to speed up 9-card rollout

For each pair of buckets in the fourth round, we need to compute the expected number of wins, losses, and draws for hands randomly drawn from those buckets. The straightforward approach of generating all $\binom{52}{2}\binom{50}{2}\binom{48}{3}\binom{45}{1}\binom{44}{1} \approx 5.56 \cdot 10^{13}$ possible ways the cards can be dealt would require more than a month of CPU time. Furthermore, this computation would have to be started from scratch when we consider a new, different abstraction. But since we know that the indexing scheme will do the "suit renaming" anyway, we are able to just generate cards for all possible indices, with a weight indicating how many symmetric situations the current cards are representing. Furthermore, we use the fact that the two sets of hole cards are symmetric to only generate those where Player 1's cards are "less" than Player 2's cards, using an arbitrary ordering of the hole cards. Doing all this gives us more than a factor 44 speed up of the 9-card rollout, bringing it down to less than a day.

Computing equilibrium strategies for the holistic abstracted four-round model

Once the abstraction has been computed, the difficult problem of computing equilibrium strategies for the abstracted game remains. The existing game-theory based players (*GS1*, *GS2*, and *Sparbot*) computed strategies by first splitting the game into two phases, and then solving the phases separately and then gluing together the separate solutions. In particular, *GS1* considers rounds 1 and 2 in the first phase, and rounds 3 and 4 in the second phase. *GS2* considers round 1, 2, and 3 in the first phase, and rounds 3 and 4 in the second phase. *Sparbot* considers rounds 1, 2, and 3 in the first phase, and rounds 2, 3, and 4 in the second phase. These approaches allow for finer-grained abstractions than what would be possible if a single, monolithic four-round model were used. However, the equilibrium finding algorithm used in each of those players was based on standard algorithms for LP that do not scale to a four-round model (except possibly for a trivially coarse abstraction).

Solving the (two) different phases separately causes important strategic errors in the player (in addition to those caused by lossy abstraction). First, it will play the first phase of the game inaccurately because it does not properly consider the later stages (the second phase) when determining the strategy for the first phase of the game. Second, it does not accurately play the second phase of the game because strategies for the second phase are derived based on beliefs at the end of the first phase, which are inaccurate.

Therefore, we want to solve for equilibrium while keeping the game in one holistic phase. To our knowledge, this is the first time this has been done. Using a holistic four-round model makes the equilibrium computation a difficult problem, particularly since our abstraction is very fine grained. As noted earlier, standard LP solvers (like CPLEX's simplex method and CPLEX's interior-point method) are insufficient for solving such a large problem. Instead, we used an implementation of Nesterov's *excessive gap technique* algorithm (Nesterov 2005), which was recently specialized for two-person zero-sum sequential games of imperfect information (Hoda, Gilpin, & Peña 2006). This algorithm is a gradient-based algorithm that requires $O(1/\epsilon)$ iterations to compute an ϵ-equilibrium, that is, a strategy for each player such that his incentive to deviate to another strategy is at most ϵ. This algorithm is an anytime algorithm since at every iteration it has a pair of feasible solutions, and the ϵ does not have to be fixed in advance. After 24 days of computing on 4 CPUs running in parallel, the algorithm had produced a pair of strategies with $\epsilon = 0.027$ small bets.

Experiments

We tested our player, *GS3*, against seven prior programs: *BluffBot*, *GS2*, *Hyperborean*,[8] *Monash-BPP*, *Spar-*

[8]There are actually two versions of *Hyperborean*: *Hyperborean-Bankroll* and *Hyperborean-Series*. The differ-

Opponent	# hands played	*GS3*'s win rate	Empirical standard deviation	95% confidence interval
Always Call	50,000	0.532	4.843	[0.490, 0.575]
Always Raise	50,000	0.442	8.160	[0.371, 0.514]
BluffBot	20,000	0.148	1.823	[0.123, 0.173]
GS2	25,000	0.222	5.724	[0.151, 0.293]
Hyperborean-Bankroll	20,000	0.099	1.779	[0.074, 0.124]
Hyperborean-Series	20,000	0.071	1.812	[0.045, 0.096]
Monash-BPP	20,000	0.669	2.834	[0.630, 0.709]
Sparbot	200,000	0.033	5.150	[0.010, 0.056]
Teddy	20,000	0.419	3.854	[0.366, 0.473]

Table 1: Experiments against static opponents. The win rate is the average number of small bets *GS3* won per hand. (The win rate against an opponent that always folds is 0.75.) *GS3* beats each opponent by a statistically significant margin.

bot, *Teddy*, and *Vexbot*. To our knowledge, this collection of opponents represents the "best of breed" in heads-up limit Texas Hold'em computer poker players. It includes all competitors from the 2006 AAAI Computer Poker Competition.

We also tested *GS3* against two (self-explanatory) benchmark strategies: *Always Call* and *Always Raise*. Although these last two strategies are completely predictable, it has been pointed out that it is important to evaluate a player against a wide range of opponents (Billings *et al.* 2003).

Experiments against static opponents

BluffBot, *GS2*, *Hyperborean*, *Monash-BPP*, *Sparbot*, and *Teddy* are static players, that is, each of them uses a mixed strategy that does not change over time.[9] Of course, *Always Call* and *Always Raise* are also static strategies.

Table 1 summarizes our experiments comparing *GS3* with the eight static opponents. One interpretation of the last column is that if zero is strictly below the interval, we can reject the null hypothesis "*GS3* is not better than the opponent" at the 95% certainty level. Thus, *GS3* beat each of the opponents with statistical significance.

The matches against *Always Call* and *Always Raise* were conducted within *Poker Academy Pro*, a commercially available software package that facilitates the design of and experimentation with poker-playing programs. We played these two strategies against *GS3* for 50,000 hands. Unsurprisingly, *GS3* beat these simple strategies very easily.

The matches against *GS2* and *Sparbot* were also conducted within *Poker Academy Pro*. *GS3* outplayed its predecessor, *GS2*, by a large margin. *Sparbot* provided *GS3* with the toughest competition, but *GS3* beat it, too, with statistical significance.

The matches against the other participants of the 2006 AAAI Computer Poker Competition beyond *GS2* (*BluffBot*, *Hyperborean*, *Monash-BPP*, and *Teddy*) were conducted on the benchmark server available for participants of that competition. One advantage of this testing environment is that it allows for *duplicate matches*, in which each hand is played twice with the same shuffle of the cards and the players' positions reversed. (Of course, the player's memories are reset so that they do not know that the same hand is being played a second time.) This reduces the role of luck, so the empirical standard deviation is lower than it would be in a normal match. Each match against these four players consisted of 20,000 duplicate hands (40,000 total). An additional way of evaluating the players in the AAAI competition is to split the experiment for each pair of competitors into 20 equal-length series, and declare as the winner of the pair the player who wins a larger number of the 20 series. Under that measure, *GS3* beat each of the opponents 20-0, except for *Hyperborean-Bankroll*, which *GS3* beat 19-1, and *Hyperborean-Series*, which *GS3* best 16-4.

Experiments against *Vexbot*

Vexbot does not employ a static strategy. It records observations about its opponents' actions, and develops a model of their style of play. It continually refines its model during play and uses this knowledge of the opponent to try to exploit his weaknesses (Billings *et al.* 2004), and is "the strongest poker program to date, having defeated every opponent it has faced" (Billings 2006; Billings & Kan 2006). (*Vexbot* did not compete in the 2006 AAAI Computer Poker Competition.)

Since *Vexbot* is remembering (and exploiting) information from each hand, the outcomes of hands in the same match are not statistically independent. Also, one known drawback of *Vexbot* is that it is possible for it to get stuck in a local minimum in its learned model (Billings 2006; Billings & Kan 2006). Hence, demonstrating that *GS3* beats *Vexbot* in a single match (regardless of the number of hands played) is not significant since it is possible that *Vexbot* happened to get stuck in such a local minimum. Therefore, instead of statistically evaluating the performance of *GS3* on a hand-by-hand basis as we did with the static players, we evaluate *GS3* against *Vexbot* on a match-by-match basis.

We performed 20 matches of *GS3* against *Vexbot*. The design of each match was extremely conservative, that is, generous for *Vexbot*. Each match consisted of 100,000 hands, and in each match *Vexbot* started with its default model of the opponent. We allowed it to learn throughout the 100,000 hands in each match (rather than flushing its memory every so often as is customary in computer poker competitions). This number of hands is many more than would actually be played between two players in practice. For example, the number of hands played in each match in the 2006 AAAI Computer Poker Competition was only 1,000.

The match results are summarized in Table 2. In every

ences between those two players are not publicly available.

[9] Since no information about *Bluffbot*, *Hyperborean*, and *Teddy* is publicly available, we are statistically evaluating *GS3*'s performance against them as though they were static.

match, *GS3* beat *Vexbot* by a large margin, with a mean win rate of 0.142 small bets per hand. The 95% confidence interval for the overall win rate is [0.133, 0.151].

One criticism that could possibly be made against the experimental methodology described above is that we did not allow *Vexbot* to learn for some period before we started recording the winnings. With this in mind, we also present (in the third column of Table 2) *GS3*'s win rate over the last 10,000 hands only, which illustrates how well *GS3* would perform if we allowed *Vexbot* to train for 90,000 hands before recording any win/loss information. As can be seen from the data, *GS3* still outperforms *Vexbot*, winning 0.147 small bets per hand on average, with a 95% confidence interval of [0.115, 0.179].

Match #	**Small bets *GS3* won per hand**	
	Over 100k hands	Over final 10k hands
1	0.129	0.197
2	0.132	0.104
3	0.169	0.248
4	0.139	0.184
5	0.130	0.150
6	0.153	0.158
7	0.137	0.092
8	0.147	0.120
9	0.120	0.092
10	0.149	0.208
11	0.098	0.067
12	0.153	0.248
13	0.142	0.142
14	0.163	0.169
15	0.165	0.112
16	0.163	0.172
17	0.108	-0.064
18	0.180	0.255
19	0.147	0.143
20	0.118	0.138
Mean:	0.142	0.147
Std. dev:	0.021	0.073
95% CI:	[0.133, 0.151]	[0.115, 0.179]

Table 2: Experiments against *Vexbot*. The third column reports *GS3*'s win rate over 10,000 hands after *Vexbot* is allowed to train for 90,000 hands.

Conclusions and future research

We presented a potential-aware automated abstraction technique for sequential imperfect information games. We also presented a custom indexing scheme based on suit isomorphisms that enables one to work on significantly larger models than was possible before.

We applied these to heads-up limit Texas Hold'em poker, and solved the abstracted game using a variant of the excessive gap technique. This is, to our knowledge, the first time that all four betting rounds have been abstracted and game-theoretically analyzed in one run (rather than splitting the game into phases). The resulting player, *GS3*, beats *BluffBot*, *GS2*, *Hyperborean*, *Monash-BPP*, *Sparbot*, *Teddy*, and *Vexbot*, each with statistical significance. To our knowledge, those competitors are the best prior programs for the game.

In the future, we would like to prove how close to optimal *GS3* is, and to experiment with the tradeoff of finer abstraction versus quality (gap) in equilibrium solving.

Acknowledgments

We gratefully thank the anonymous reviewers for their suggestions for improving the experimental methodology. We would also like to thank Christian Smith and Martin Zinkevich for their efforts in enabling our experiments on the benchmark server at the University of Alberta.

References

Billings, D., and Kan, M. 2006. A tool for the direct assessment of poker decisions. *ICGA Journal* 29(3):119–142.

Billings, D.; Davidson, A.; Schaeffer, J.; and Szafron, D. 2002. The challenge of poker. *Artificial Intelligence* 134(1-2):201–240.

Billings, D.; Burch, N.; Davidson, A.; Holte, R.; Schaeffer, J.; Schauenberg, T.; and Szafron, D. 2003. Approximating game-theoretic optimal strategies for full-scale poker. In *Proc. of the Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 661–668.

Billings, D.; Bowling, M.; Burch, N.; Davidson, A.; Holte, R.; Schaeffer, J.; Schauenberg, T.; and Szafron, D. 2004. Game tree search with adaptation in stochastic imperfect information games. In *Proc. of the Int. Conf. on Computers and Games (CG)*, 21–34.

Billings, D. 2006. *Algorithms and Assessment in Computer Poker*. Ph.D. Dissertation, University of Alberta.

Bollobás, B. 1986. *Combinatorics*. Cambridge University Press.

Gilpin, A., and Sandholm, T. 2006a. A competitive Texas Hold'em poker player via automated abstraction and real-time equilibrium computation. In *Proc. of the National Conf. on Artificial Intelligence (AAAI)*.

Gilpin, A., and Sandholm, T. 2006b. Finding equilibria in large sequential games of imperfect information. In *ACM Conference on Electronic Commerce (ACM-EC)*, 160–169.

Gilpin, A., and Sandholm, T. 2007. Better automated abstraction techniques for imperfect information games, with application to Texas Hold'em poker. In *Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS)*.

Hoda, S.; Gilpin, A.; and Peña, J. 2006. A gradient-based approach for computing Nash equilibria of large sequential games. Manuscript. Presented at INFORMS-06.

Koller, D., and Megiddo, N. 1992. The complexity of two-person zero-sum games in extensive form. *Games and Economic Behavior* 4(4):528–552.

Korb, K.; Nicholson, A.; and Jitnah, N. 1999. Bayesian poker. In *Proc. of the Conf. on Uncertainty in AI (UAI)*, 343–350.

Littman, M., and Zinkevich, M. 2006. The 2006 AAAI Computer-Poker Competition. *ICGA Journal* 29(3):166.

Miltersen, P. B., and Sørensen, T. B. 2007. A near-optimal strategy for a heads-up no-limit Texas Hold'em poker tournament. In *Int. Conf. on Autonomous Agents and Multi-Agent Systems*.

Nesterov, Y. 2005. Excessive gap technique in nonsmooth convex minimization. *SIAM Journal of Optimization* 16(1):235–249.

Romanovskii, I. 1962. Reduction of a game with complete memory to a matrix game. *Soviet Mathematics* 3:678–681.

Sklansky, D. 1999. *The Theory of Poker*. Two Plus Two Publishing, fourth edition.

Sturtevant, N.; Zinkevich, M.; and Bowling, M. 2006. Prob-max^n: Opponent modeling in n-player games. In *Proc. of the National Conf. on Artificial Intelligence (AAAI)*, 1057–1063.

von Stengel, B. 1996. Efficient computation of behavior strategies. *Games and Economic Behavior* 14(2):220–246.